

***TMS320C6000 Imaging Developer's Kit
(IDK)
Video Device Driver
User's Guide***

Literature Number: SPRU499
December 2000



IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

Read This First

About This Manual

The Imaging Developer's Kit (IDK), is a complete system consisting of hardware and software designed to demonstrate the capabilities of TI's TMS320C6000 family of DSPs in the field of video/image processing. The IDK also serves as a rapid prototyping platform for the development of image and video processing algorithms. This document describes the software and hardware components provided in the IDK.

How to Use This Manual

This document contains the following chapters:

- Chapter 1 – Overview**, provides information about the software module block diagram, software architecture, and describes how to use video device drivers.
- Chapter 2 – Video Display System**, provides a block diagram and describes the function of the video display system. Also contains a video display API reference (VDIS).
- Chapter 3 – Video Capture System**, explains and illustrates the video display subsystem.
- Chapter 4 – Examples**, provides code examples of video device drivers' uses.

Related Documentation From Texas Instruments

The following references are provided for further information:

Documentation

TMS320C6000 Imaging Developer's Kit (IDK) User's Guide (Literature number SPRU494)

TMS320C6000 Imaging Developer's Kit (IDK) Programmer's Guide
(Literature number SPRU495)

C6000 JPEG Information:

- TMS320C6000 JPEG Implementation Application Report* (Literature number SPRA704)
- Optimizing JPEG on the TMS320C6211 With 2-Level Cache Application Report* (Literature number SPRA705)

C6000 H.263 Information:

- H.263 Decoder: TMS320C6000 Implementation Application Report* (Literature number SPRA703)
- H.263 Encoder: TMS320C6000 Implementation Application Report* (Literature number SPRA721)

Text Conventions

The following typographical conventions are used in this specification:

- Text inside back-quotes (“”) represents pseudo-code
- Program source code, function and macro names, parameters, and command line commands are shown in a mono-spaced font.

Contents

1	Overview	1-1
1.1	Software Module Block Diagram	1-2
1.2	Using the Driver Library	1-3
1.3	Software Architecture	1-4
2	Video Display System	2-1
2.1	Display Block Diagram	2-2
2.2	Display Timing and Events	2-3
2.3	Triple Buffering Scheme	2-4
2.4	Using the Display APIs	2-5
2.5	Display Buffer Format	2-7
2.6	Video Display API Reference (VDIS)	2-8
3	Video Capture System	3-1
3.1	Capture Block Diagram	3-2
3.2	Capture Timing and Events	3-3
3.3	Triple Buffering Scheme	3-4
3.4	Using the Capture APIs	3-5
3.5	Capture Buffer Format	3-7
3.6	Video Capture API Reference (VCAP)	3-9
4	Examples	4-1
4.1	Draw a Box in 640x480x16 Display Mode	4-2
4.2	Draw a Box in 640x480x8 Display Mode	4-3
4.3	NTSC Capture to 640x480x16 Display Loopback	4-4
4.4	NTSC Capture to 640x480x8 Display Loopback	4-6

Figures

1-1	Software Architecture	1-5
2-1	Video Display Subsystem Block Diagram	2-2
2-2	Display Buffer Format	2-7
3-1	Video Capture Subsystem Block Diagram	3-2
3-2	Capture Buffer Format	3-8

Overview

The Imaging Developer's Kit, or IDK, is a complete system consisting of hardware and software designed to demonstrate the capabilities of TI's TMS320C6000 family of DSPs in the field of video/image processing. A key component of this system is the video device drivers that offer a simple API for accessing the video hardware. This document describes these drivers and how to use them.

Topic	Page
1.1 Software Module Block Diagram	1-2
1.2 Using the Driver Library	1-3
1.3 Software Architecture	1-4

1.1 Software Module Block Diagram

The video hardware has a video capture subsystem with a TVP5022 video decoder chip and a video display subsystem with a TVP3026 RAMDAC. There are two independent APIs, one for video display (VDIS module) and one for video capture (VCAP module). The drivers depend on DSP/BIOS II and take advantage of the tasking model using semaphores. The drivers also use the chip support library (CSL) for all on-chip DSP peripheral access.

The driver software is written specifically for the imaging daughter card designed by TI. However, the APIs are simple enough that it may possible to port the driver APIs to other hardware platforms.

The display driver uses triple buffering and these buffers are stored in the DSK's SDRAM. The display hardware generates line events that are tied to an EDMA channel that in turn transfers lines of video from the display buffers to the display hardware. This display data is sent to the RAMDAC and output onto a standard 15-pin VGA monitor connector. The display hardware also generates a frame interrupt to the DSP every vertical sync. The display driver processes this interrupt and uses it to synchronize with the application using a DSP/BIOS semaphore.

The daughter card has dedicated SDRAM for video capture data and triple buffering is used. Composite video input goes into a video decoder chip that in turn is written into the capture RAM. The capture RAM is memory-mapped into the DSPs address space (read-only). This allows the DSP to directly read the frames of captured video. The capture hardware also generates frame events that trigger an interrupt to the DSP every vertical sync. The capture driver processes this interrupt and uses it to synchronize with the application using a DSP/BIOS semaphore.

1.2 Using the Driver Library

You need to do the following to use the drivers:

- Include the driver header files
 - `#include <vdis.h>` to use display driver
 - `#include <vcap.h>` to use capture driver
- Link in the driver library
 - `-l vcard.lib`
- Predefine processor ID symbol, required for CSL (see CCS project options under Compiler→Preprocessor).
 - `-d CHIP_6711`
- Link in the CSL library file
 - `-l csl6711.lib`

There are a couple basic steps every application needs to do to use the driver APIs. Open a device (capture or display), call the device APIs, then close it when done. Once a device is open, the APIs may be called over and over again, you only close a device when completely done with it. Generally, an application will open the devices at the beginning and leave them open indefinitely.

- Open device:
 - `VDIS_open()`
 - `VCAP_open()`
- Call device APIs:
 - `VDIS_config()`
 - `VDIS_toggleBufs()`
 - `VDIS_fill()`
 - `VCAP_config()`
 - `VCAP_getFrame()`
- Close the devices (usually not done):
 - `VDIS_close()`
 - `VCAP_close()`

The open functions are used to do certain initialization and allocates system resources such as EDMA channels. Closing a device frees up any resources allocated during open.

1.3 Software Architecture

The device driver software architecture is illustrated in Figure 1–1. It shows the different modules and the interfacing between them. The *VCARD* block represents the capture and display drivers. The top-level modules for the drivers are *VDIS* and *VCAP*. The *VDIS* module defines the video display API and the *VCAP* module defines the video capture API.

The *VDIS* module interfaces into the *TVP2036* submodule which is responsible for configuring the *RAMDAC*.

The capture side is a little more complex; the *VCAP* module interfaces to the *TVP5022* video decoder module that in turn interfaces to the *I2C* module and two micro-code modules.

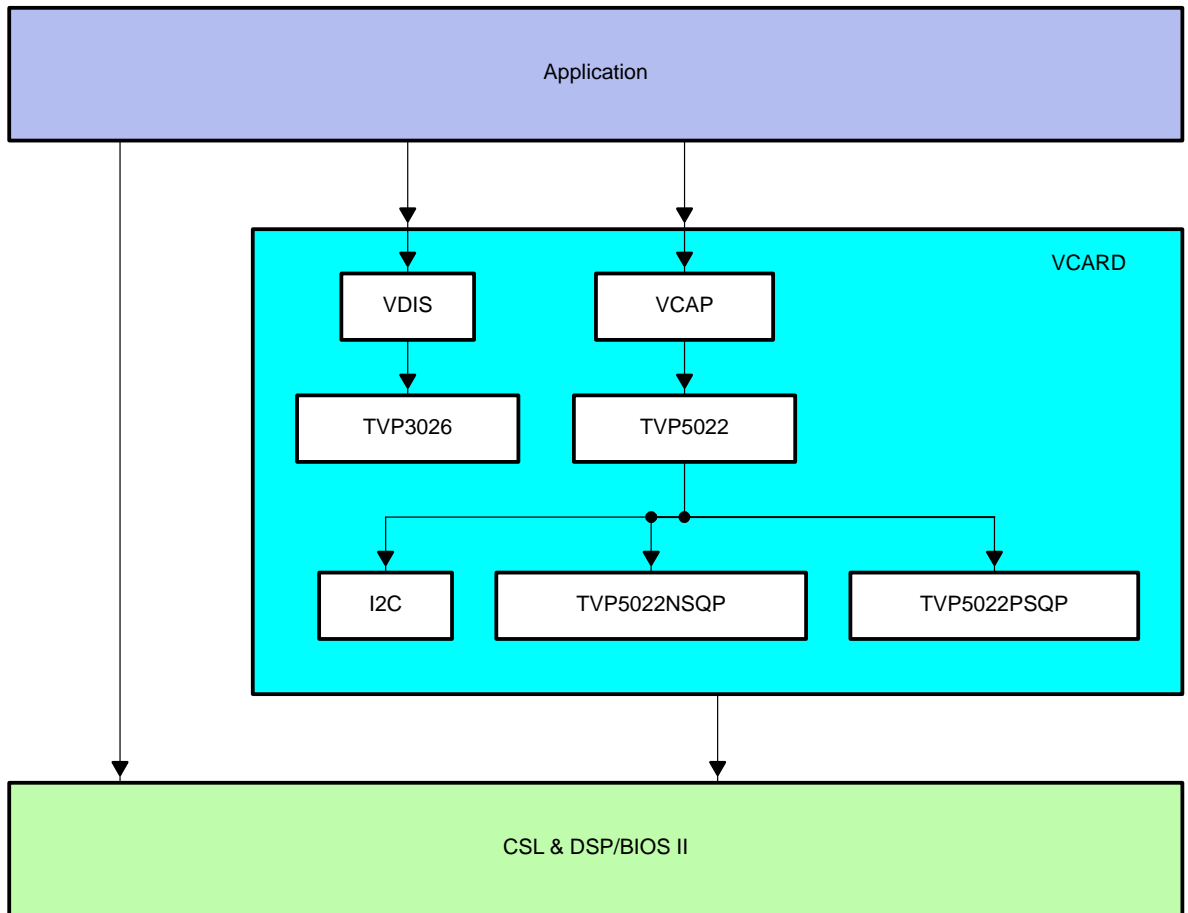
All modules potentially interface into the *CSL* and *DSP/BIOS*. At the top is the application. It interfaces into the *VCAP* and *VDIS* module and of course may interface into the *CSL* and *DSP/BIOS*.

The driver software is written entirely in C. The modules consist of the following files:

- ❑ **VDIS**: *vdis.h*, *vdis.c*
- ❑ **VCAP**: *vcap.h*, *vcap.c*
- ❑ **TVP3026**: *tv3026.h*, *tv3026.c*
- ❑ **TVP5022**: *tv5020.h*, *tv5020.c*
- ❑ **I2C**: *i2c.h*, *i2c.c*
- ❑ **TVP5022NSQP**: *tv5020nsqp.h*, *tv5020nsqp.c*
- ❑ **TVP5022PSQP**: *tv5020psqp.h*, *tv5020psqp.c*

Please note that while the code file names above are with reference to *TVP5020*, they are equally valid for *TVP5022*. All of these source files are located in the source archive ***vcard.src***.

Figure 1–1. Software Architecture



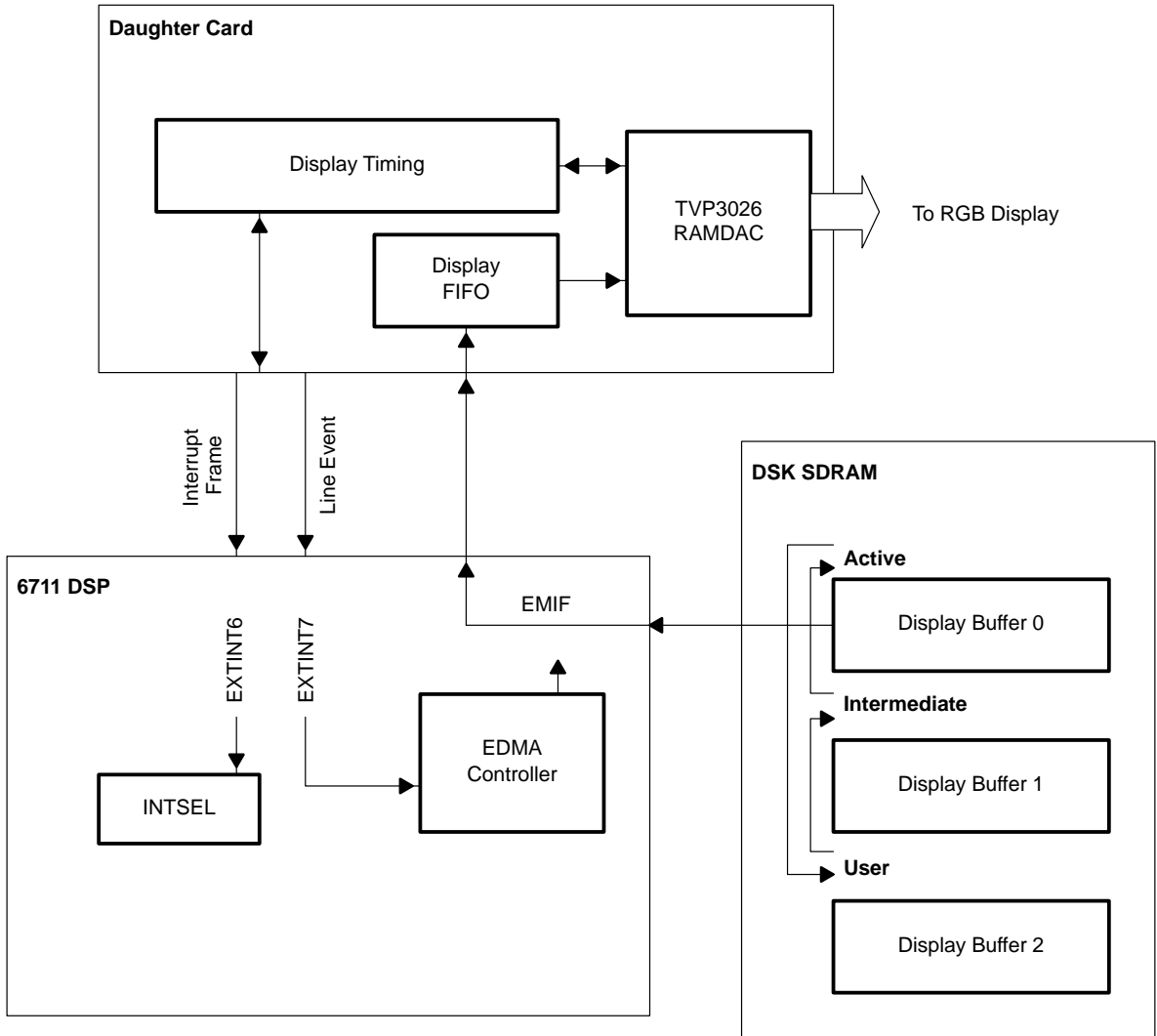
Video Display System

Topic	Page
2.1 Display Block Diagram	2-2
2.2 Display Timing and Events	2-3
2.3 Triple Buffering Scheme	2-4
2.4 Using the Display APIs	2-5
2.5 Display Buffer Format	2-7
2.6 Video Display API Reference (VDIS)	2-8

2.1 Display Block Diagram

Figure 2–1 shows the block diagram of the video display subsystem. First of all, three display buffers are allocated in .bss which gets linked into the DSK’s SDRAM. These buffers are allocated for the worst case, that is 800x600x16 = 960,000 bytes each times 3 = 2,880,000 bytes total. When a lower resolution is used, some of the space is left unused.

Figure 2–1. Video Display Subsystem Block Diagram



2.2 Display Timing and Events

The *Timing* block of the diagram is implemented in an FPGA and generates the display HSYNC and VSYNC signals based on programmable parameters. VSYNC is connected to EXTINT6 and the HSYNC is connected to EXTINT7. EXTINT6 is configured to generate a CPU interrupt and the interrupt service routine is defined in the video driver, VDIS_isr(). This ISR does nothing more than post the display semaphore which is used by the VDIS_toggleBufs() function to wait for new frames.

EXTINT7 is configured to trigger an EDMA event that copies one line of display data from the current display buffer to the daughter card (FIFO). This line of video then gets displayed through the TVP3026. The EDMA parameters are setup to autoincrement to the next line then after a whole frame is displayed, the parameters reset to the beginning of the buffer again. These EDMA operations happen at a fairly high rate up to around 40,000 transfers per second and transfer up to 1600 bytes each so they are submitted at a high priority (see options field of EDMA parameters).

Note:

Since the display driver depends on DSP/BIOS and interrupts are used, it is important that the display interrupt service routine VDIS_isr() is configured into HWI EXTINT6 in the DSP/BIOS configuration tool. Also, the “Use Dispatcher” box must be checked. If this is not done, the display interrupt will never get called and the display semaphore will never get posted.

2.3 Triple Buffering Scheme

A triple buffering scheme is used for the display such that the application can always get a new buffer without waiting, if so desired. It works like this, one of the buffers is *active* meaning that the EDMA events are currently moving data from that buffer to the display hardware. A second buffer is owned by the application (*user*), it's the one your application is currently rendering into. The third buffer is the next one the application will get upon calling `VDIS_toggleBufs()`, (*intermediate*). As you can see, having three buffers allows the application to obtain a buffer without waiting for a new frame. If there were only two buffers, the swap could only occur on frame boundaries which would mean waiting.

If the application attempts to receive buffers faster than they can be displayed (60Hz for example) then some frames will be missed. This can happen if you call `VDIS_toggleBufs(0)` which basically gets the next buffer without waiting for a new one to become available. If you call this faster than the display rate, then the function will swap between the *user* buffer and the *intermediate* buffer over and over again. Hence, you may end up losing a frame but no harm is done.

On the other hand, if the application requests buffers at a rate slower than the display rate, then some frames will get displayed repeatedly. This is because the *active* buffer will only change if the user called `VDIS_toggleBufs()` since the last new frame came in.

2.4 Using the Display APIs

Once the display is opened and configured, the application only has to call `VDIS_toggleBufs()` to obtain a new buffer to render into. This function takes one argument, *timeout* that determines how to wait. Possible values are 0, `n`, or `SYS_FOREVER`. If 0 is used, the function will not wait at all and immediately return the next buffer. If `n` (an integer value) is used, then the function will wait for at least that many system ticks (see DSP/BIOS users guide) for a new frame to become available. Then the next frame is returned regardless if a new one came in or not. Specifying `SYS_FOREVER` causes the function to wait indefinitely for a new frame to arrive. This waiting is done by blocking on a built-in semaphore, there is no *spinning*. This semaphore is posted once every vertical sync by the interrupt service routine. If you want your application to be synchronized with the display, use `VDIS_toggleBufs(SYS_FOREVER)`. If your application is synchronized some other way (with capture for example), then use `VDIS_toggleBufs(0)`.

Note:

Since `VDIS_toggleBufs()` blocks on a semaphore, it is important that this function ONLY be called from the context of a DSP/BIOS task. This means don't call this function from *main()*.

Before calling any of the VDIS functions, you must first open the display by calling `VDIS_open()`. This function performs some internal initialization and allocates system resources, i.e. EDMA channel, semaphore, etc. If your application ever finishes with the display, then these resources may be freed up by calling `VDIS_close()`.

Once the display is opened, it must be configured by calling `VDIS_config(mode)` where *mode* is the desired display mode. Currently, the driver supports these modes:

- VDIS_640X480X8**: 640x480 8-bit greyscale @ 60Hz
- VDIS_640X480X16**: 640x480 16-bit color (565 packed format) @ 60Hz
- VDIS_800X600X8**: 800x600 8-bit greyscale @ 60Hz
- VDIS_800X600X16**: 640x480 16-bit color (565 packed format) @ 60Hz

If you ever wish to disable the display or turn it off, call `VDIS_reset()`. You may call `VDIS_fill(buffPtr, pixelValue)` if you want to fill a buffer with a pixel value. This function simply uses a CPU *for loop* to do the fill but does not comprehend the pixel depth. If you call the fill function, note that the cache is not flushed, this is up to the user.

A typical display loop looks something like this:

```
/* execute from the context of a task */
someTaskFunc() {

    /* open the display */
    if (VDIS_open()) {

        /* configure the display */
        VDIS_config(VDIS_640X480X16);

        /* infinite processing loop */
        while (1) {

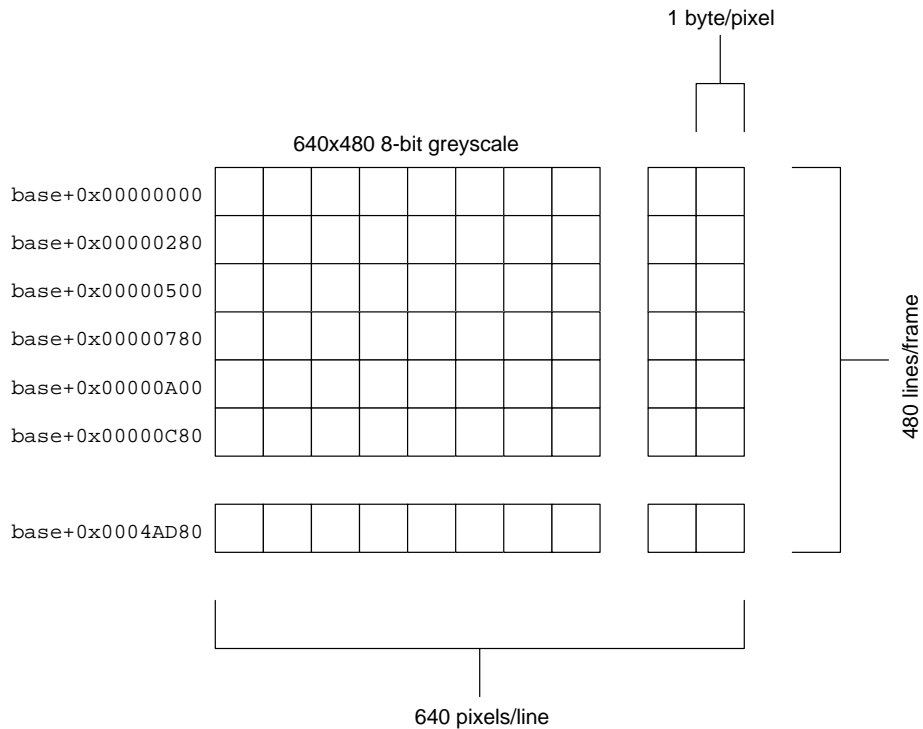
            /* get the next available display buffer */
            dispBuff = VDIS_toggleBufs(SYS_FOREVER);

            /* write data to display buffer */
            dispBuff[] = F(x);
        }
    }
}
```

2.5 Display Buffer Format

Figure 2–2 illustrates how the individual display buffers are organized. The buffer is stored continuously in memory such that the start of one line backs up against the end of the previous line. For 8-bit display modes (as in the figure), each pixel takes up one byte, for 16-bit modes, 2 bytes are used per pixel.

Figure 2–2. Display Buffer Format



2.6 Video Display API Reference (VDIS)

VDIS_close *Closes display device*

Function void VDIS_close();

Arguments none

Return Value none

Description This function closes the display device and frees up any resources allocated during VDIS_open(). The display is also reset.

Generally, this function is never called because video applications are usually infinite processing loops. Hence, the display is always left open.

Here are the steps taken.

- Check to make sure the display is open
- Call VDIS_reset()
- Close the EDMA channel
- Free the EDMA parameter tables
- Delete the DSP/BIOS semaphore
- Returns

Example `VDIS_close();`

VDIS_config *Configures and enables display hardware for mode specified*

Function int VDIS_config(
int mode
);

Arguments mode Specifies the video display mode. Supported modes are:

- VDIS_640X480X8 – 640x480 8bpp greyscale @ 60Hz
- VDIS_640X480X16 – 640x480 16bpp 565 @ 60 Hz
- VDIS_800X600X8 – 800x600 8bpp greyscale @ 60Hz
- VDIS_800X600X16 – 800x600 16bpp 565 @60Hz

Return Value success Returns 0 on failure.

Description This function configures and enables the display hardware for the mode specified. The display must be opened before calling this function. See `VDSI_open()`. Here are the steps taken:

- Check to make sure display is open
- Calls `VDIS_reset()`
- Updates `VDIS_settings` structure
- Sets up FPGA timing parameters
- Configures the TVP3026
- Configures the EDMA
- Configures the display interrupt
- Enables everything
- Synchronizes for two frames
- Returns

Example `VDIS_config(VDIS_640X480X16);`

VDIS_fill

Fills buffer with specified pixel value

Function

```
void VDIS_fill(
    void *buff,
    Uint32 pixel
);
```

Arguments

<code>buff</code>	Pointer to display buffer to fill, usually obtained by a call to <code>VDIS_toggleBufs()</code> .
<code>pixel</code>	The pixel value to fill with. Depends on display mode, for 8 bit mode, only lower 8-bits are used. For 16-bit mode, lower 16-bits are used.

Return Value none

Description This function fills a buffer with a specified pixel value. The buffer is assumed to have the dimensions defined in `VDIS_settings`. The fill is performed using a CPU *for loop* and if the current display mode is 8 bpp, then 8-bit CPU stores are performed. If 16 bpp, then 16-bit stores are performed. This function in no way attempts to flush the cache when done with the fill operation. So keep in mind that after calling `VDIS_fill()`, some of the fill data may still be sitting in cache. It is up to the user to flush the cache when appropriate.

Also note that filling a display buffer does not mean it will be displayed right away. You still have to cycle that buffer until it is *active* by calls to `VDIS_toggleBufs()`.

Example 1

```
/* this example will fill a greyscale display with */
/* light grey                                     */

VDIS_open();
VDIS_config(VDIS_640X480X8);

/* to prime up all three display buffers */
for (x=0; x<3; x++) {
    buff = VDIS_toggleBufs(SYS_FOREVER);
    VDIS_fill(buff,0x80);
}
```

Example 2

```
/* this example will fill a color display with */
/* bright blue                                   */

VDIS_open();
VDIS_config(VDIS_640X480X16);

/* to prime up all three display buffers */
for (x=0; x<3; x++) {
    buff = VDIS_toggleBufs(SYS_FOREVER);
    VDIS_fill(buff,0x001F);
}
```

VDIS_isr

Built-in interrupt service routine for display driver

Function int VDIS_isr();

Arguments none

Return Value none

Description This is the built-in interrupt service routine for the display driver. The user should never call this function directly. It is exported globally so that it can be referenced in the interrupt service table.

When using DSP/BIOS, HWI EXTINT6 must be set to call this function and the "Use Dispatcher" box MUST be checked.

VDIS_open *Opens display device*

Function	int VDIS_open();
Arguments	none
Return Value	success Returns 0 on failure. Failure could happen if required system resources could not be allocated.
Description	<p>This function opens the display device and must be called before calling any other display APIs. Generally, this function is called only once at the beginning of your program. VDIS_reset() is called and system resources are allocated. To free up these resources, call VDIS_close().</p> <p>Here are the steps taken.</p> <ul style="list-style-type: none"> <input type="checkbox"/> Check to make sure the display is not already open <input type="checkbox"/> Open EDMA channel <input type="checkbox"/> Allocate EDMA reload parameter tables <input type="checkbox"/> Dynamically create a DSP/BIOS semaphore <input type="checkbox"/> Call VDIS_reset() <input type="checkbox"/> Returns
Example	<pre>success = VDIS_open();</pre>

VDIS_reset *Resets display hardware*

Function	int VDIS_reset();
Arguments	none
Return Value	none
Description	<p>This function resets the display hardware which in effect turns it off. Here are the steps taken:</p> <ul style="list-style-type: none"> <input type="checkbox"/> Check to make sure display is open <input type="checkbox"/> Disables EDMA channel <input type="checkbox"/> Clears all FPGA timing parameters <input type="checkbox"/> Fills all display buffers with 0 <input type="checkbox"/> Updates VDIS_settings structure <input type="checkbox"/> Disables and clears the display CPU interrupt <input type="checkbox"/> Returns
Example	<pre>VDIS_reset();</pre>

VDIS_settings *Global variable exported out of VDIS module*

Global Variable	extern far VDIS_Settings VDIS_settings;	
Members	mode	Current mode, i.e. VDIS_640X480X16
	hres	Horizontal resolution in pixels, i.e. 640
	vres	Vertical resolution in pixels, i.e. 480
	bpp	Bits per pixel, i.e. 16
	pitch	Display pitch, number of bytes from start of one line to the start of the next line, i.e. 640 for 640x480x8
	buffsz	Total size of one display buffer in bytes
	fps	Display rate in frames per second, i.e. 60

Description This is a global variable exported out of the VDIS module that contains information about the current display mode. Its purpose is to make it easy for the application code to read this information. It is useful when the application code needs react differently depending on display settings.

Note: Do **not** write to any members of this structure, it is intended to be read-only.

This structured is altered by VDIS_open(), VDIS_config(), and VDIS_reset(). Generally, the user calls VDIS_config() then uses information from this structure.

Example `VDIS_config(VDIS_640X480X16);`

```
dx = VDIS_settings.hres;  
dy = VDIS_settings.vres;
```

VDIS_toggleBufs *Toggles display buffers*

Function	void *DIS_toggleBufs(int timeout);	
Arguments	timeout	Specifies how to wait for a new available display buffer. 0 – don't wait at all n – wait for n ticks SYS_FOREVER – wait for ever
Return Value	buff	Returns a pointer to a free display buffer.

Description

This function toggles the display buffers and returns a pointer to a free display buffer. You have a choice of how you want to wait for the new buffer to become available. Specifying 0 means don't wait at all. This will return the current available buffer regardless if a new display event has occurred since the last time you called this function. Specifying a positive integer causes this function to *block* on a semaphore for *n* system ticks until a new display event occurs. If a new event does not occur by the time *n* ticks expires, then the most current free buffer is returned. Specifying SYS_FOREVER causes this function to *block* on a semaphore until an event occurs.

The display hardware generates an interrupt on every vertical sync event. The interrupt service routine posts the display semaphore.

The return value is a pointer to a free display buffer. Once you have this pointer, you are expected to fill the buffer up with display data. You exclusively own this buffer until you call this function again. At that time, this buffer gets displayed.

This function performs these basic steps:

- Links the EDMA to the next display buffer (*intermediate*)
- Pends on the display semaphore
- Increments an internal buffer index number
- Returns the new buffer pointer

Example

```

/* execute from the context of a task */
someTaskFunc() {

    /* open the display */
    if (VDIS_open()) {

        /* configure the display */
        VDIS_config(VDIS_640X480X16);

        /* infinite processing loop */
        while (1) {

            /* get the next available display buffer */
            dispBuff = VDIS_toggleBufs(SYS_FOREVER);

            /* write data to display buffer */
            dispBuff[] = F(x);
        }
    }
}

```

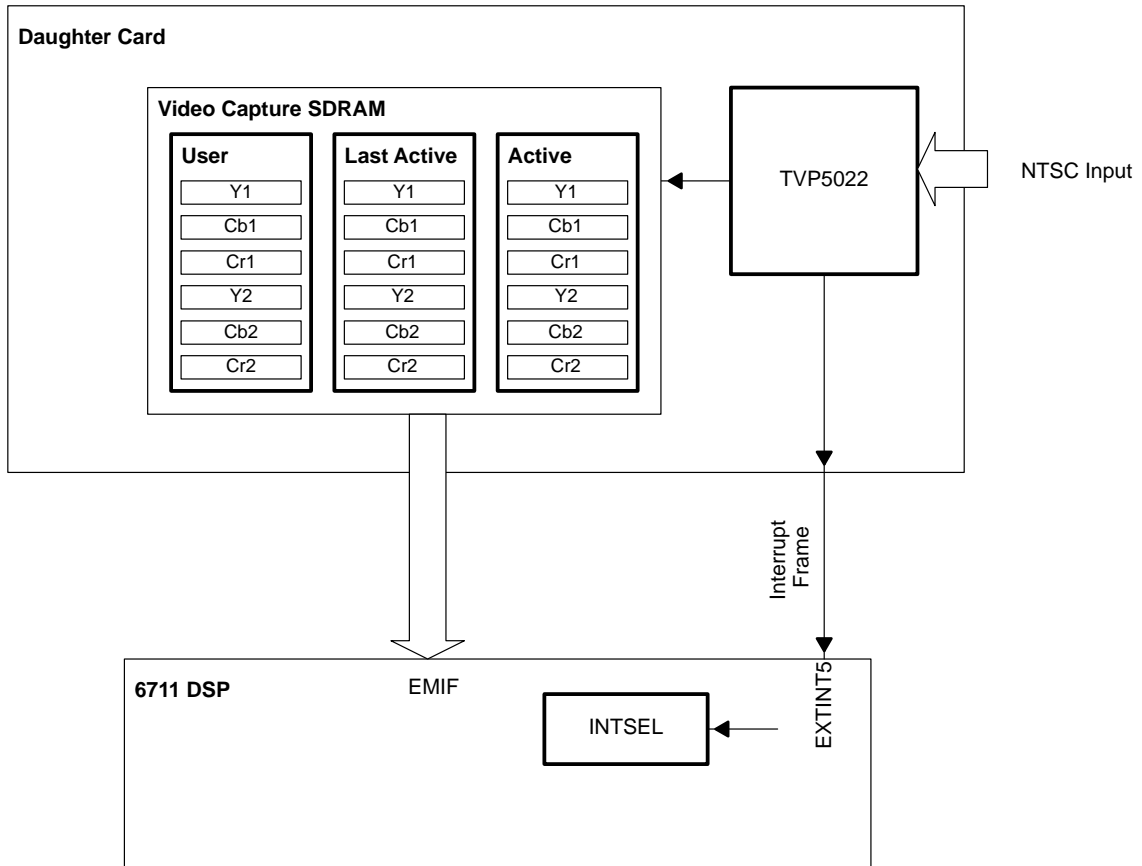

Video Capture System

Topic	Page
3.1 Capture Block Diagram	3-2
3.2 Capture Timing and Events	3-3
3.3 Triple Buffering Scheme	3-4
3.4 Using the Capture APIs	3-5
3.5 Capture Buffer Format	3-7
3.6 Video Capture API Reference (VCAP)	3-9

3.1 Capture Block Diagram

Figure 3–1 shows the block diagram of the video capture subsystem. First of all, three capture buffers are hardwired into the SDRAM of the daughter card and a triple buffering scheme is used. The daughter card is designed such that data coming in from the TVP5022 decoder chip is automatically stored in the daughter card SDRAM at the appropriate location. Software determines which of the three buffers the capture hardware stores into.

Figure 3–1. Video Capture Subsystem Block Diagram



3.2 Capture Timing and Events

One event is tied from the capture subsystem to the DSP. This is EXTINT5 triggered by every other VSYNC (once per frame which is every other field) from the TVP5022 chip and is configured to generate a CPU interrupt. The interrupt service routine is defined internal to the driver, `VCAP_isr()`. This ISR does nothing more than post the capture semaphore which is used by the `VCAP_getFrame()` function to wait for new frames.

The capture buffers are directly mapped into the DSP's address space as read-only. This means the CPU or EDMA can directly read these buffers.

Note:

Since the capture driver depends on DSP/BIOS and interrupts are used, it is important that the capture interrupt service routine `VCAP_isr()` is configured into HWI EXTINT5 in the DSP/BIOS configuration tool. Also, the "Use Dispatcher" box must be checked. If this is not done, the capture interrupt will never get called and the capture semaphore will never get posted.

3.3 Triple Buffering Scheme

A triple buffering scheme is used for the capture such that the application can always get a new buffer without waiting, if so desired. It works like this: one of the buffers is *active*, meaning that the capture hardware is currently storing data into this buffer. A second buffer is owned by the application (*user*), which is the one your application is currently reading from. The third buffer is the last one filled by the capture hardware (*last active*). Having three buffers allows the application to obtain a new buffer without waiting for a new frame and the new buffer always contains the most recent captured data. If there were only two buffers, the swap could only occur on frame boundaries which would mean waiting.

If the application attempts to grab buffers faster than they can be captured (30Hz for example) then duplicate frames will be returned. This can happen if you call `VCAP_getFrame(0)` which basically gets the next frame without waiting for a new one to become available. If you call this faster than the capture rate, then the function will continuously return the same frame until a new one comes in.

On the other hand, if the application requests buffers at a rate slower than the capture rate, then some captured frames will get lost (overwritten).

3.4 Using the Capture APIs

Once the capture is opened and configured, the application only has to call `VCAP_getFrame()` to obtain a new frame of captured data. This function takes one argument, *timeout* that determines how to wait. Possible values are 0, `n`, or `SYS_FOREVER`. If 0 is used, the function will not wait at all and immediately return the next buffer. If `n` (an integer value) is used, then the function will wait for at least that many system ticks (see DSP/BIOS users guide) for a new frame to become available. Then the next frame is returned regardless if a new one came in or not. Specifying `SYS_FOREVER` causes the function to wait indefinitely for a new frame to arrive. This waiting is done by blocking on a built-in semaphore; there is no *spinning*. This semaphore is posted once every vertical sync by the interrupt service routine. If you want your application to be synchronized with the capture, use `VCAP_getFrame(SYS_FOREVER)`. If your application is synchronized some other way (with display for example), then use `VCAP_getFrame(0)`.

Note:

Since `VCAP_getFrame()` blocks on a semaphore, it is important that this function ONLY be called from the context of a DSP/BIOS task. This means don't call this function from *main()*.

Before calling any of the VCAP functions, you must first open the capture by calling `VCAP_open()`. This function performs some internal initialization and allocates system resources, i.e. semaphore. If your application ever finishes with the capture, then these resources may be freed up by calling `VCAP_close()`.

Once the capture is opened, it must be configured by calling `VCAP_config(mode)` where *mode* is the desired capture mode. Currently, the driver supports these modes:

- VCAP_NTSC**: 640x480 YCbCr 4:2:2 square pixel @ 30 frames/second
- VCAP_PAL**: 768x576 YCbCr 4:2:2 square pixel @ 25 frames/second

If you ever wish to disable the capture or turn it off, call `VCAP_reset()`.

A typical capture loop looks something like this:

```
/* execute from the context of a task */
someTaskFunc() {

    /* open the capture */
    if (VCAP_open()) {

        /* configure the capture */
        VCAP_config(VCAP_NTSC);

        /* infinite processing loop */
        while (1) {

            /* get the next available capture frame */
            input = VCAP_getFrame(SYS_FOREVER);

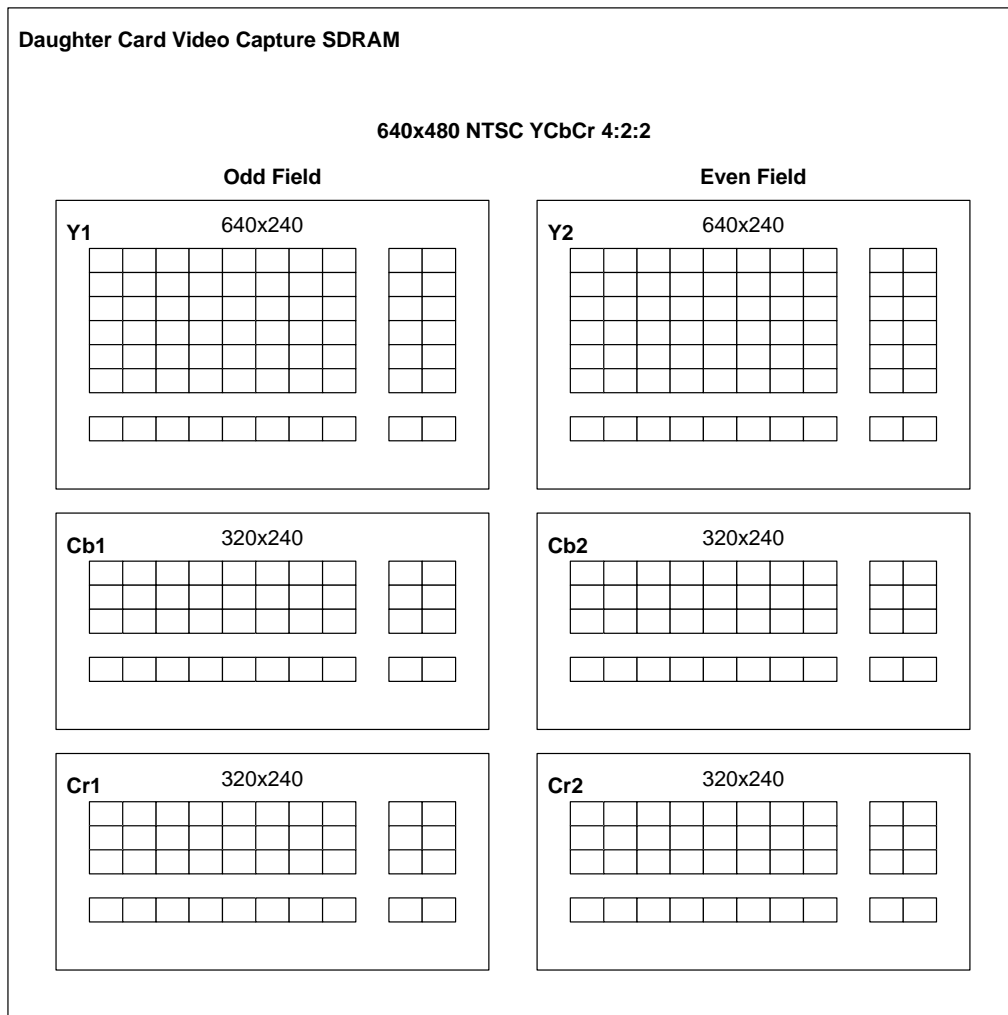
            /* process the captured data */
            x = F(input);
        }
    }
}
```

3.5 Capture Buffer Format

Figure 3–2 illustrates how the individual capture buffers are organized. There are three separate buffers used to implement the triple buffering scheme and each buffer is made up of three components, Y, Cb, and Cr. The component sub-buffers are not continuous. This means there may be gaps between the Y, Cb, and Cr buffers. Also, there are separate component buffers for the even and odd fields. All buffer (sub-buffer) addresses are fixed and hardwired. The figure shows the NTSC buffers but the same thing applies for PAL, just different dimensions.

When the user calls `VCAP_getFrame()`, the return value is a pointer to a structure that has individual members that point to each of the sub-component buffers, for both even and odd fields. It is up to the user to merge these together if they want the full interlaced frame.

Figure 3–2. Capture Buffer Format



3.6 Video Capture API Reference (VCAP)

VCAP_close *Closes capture device*

Function	<code>void VCAP_close();</code>
Arguments	none
Return Value	none
Description	<p>This function closes the capture device and frees up any resources allocated during <code>VCAP_open()</code>. The capture is also reset.</p> <p>Generally, this function is never called because video applications are usually infinite processing loops. Hence, the capture is always left open.</p> <p>Here are the steps taken.</p> <ul style="list-style-type: none"> <input type="checkbox"/> Check to make sure the capture is open <input type="checkbox"/> Call <code>VCAP_reset()</code> <input type="checkbox"/> Delete the DSP/BIOS semaphore <input type="checkbox"/> Returns
Example	<code>VCAP_close();</code>

VCAP_config *Configures and enables capture hardware*

Function	<code>int VCAP_config(int mode,);</code>
Arguments	<p>mode Specifies capture mode. Supported modes are:</p> <ul style="list-style-type: none"> <input type="checkbox"/> <code>VCAP_NTSC</code> 640x480 YCbCr 4:2:2 square pixel @ 30 fps <input type="checkbox"/> <code>VCAP_PAL</code> 768x576 YCbCr 4:2:2 square pixel @ 25 fps
Return Value	success Returns zero on failure.
Description	This function configures and enables the capture hardware for the mode specified.
Example	<code>VCAP_config(1);</code>

VCAP_getFrame Returns most recent captured frame of video data

Function VCAP_Frame *VCAP_getFrame(
int timeout
);

Arguments timeout Specifies how to wait for a new frame.
0 – don't wait at all
n – wait for n ticks
SYS_FOREVER – wait for ever

Return Value frame Returns a frame pointer to the latest captured frame. The frame object has the following fields.

- void *y1
- void *cr1
- void *cb1
- void *y2
- void *cr2
- void *cb2

Description This function returns the most recent captured frame of video data. You have a choice of how you want to wait for the new frame. Specifying 0 means don't wait at all. This will return the most recent frame captured regardless if a new frame has arrived since the last time you called this function. Specifying a positive integer causes this function to *block* on a semaphore for *n* system ticks until a new frame arrives. If a new frame does not arrive by the time *n* ticks expires, then the most recent frame is returned. Specifying SYS_FOREVER causes this function to *block* on a semaphore until a new frame arrives.

The capture hardware generates an interrupt everytime a new frame arrives. The interrupt service routine posts the capture semaphore.

The return value is a pointer to a VCAP_Frame object whose members are pointers to each component of the captured frame.

Example

```
VCAP_Frame input;  
...  
while (1) {  
    input = VCAP_getFrame(SYS_FOREVER);  
    ...  
}
```

VCAP_isr *Built-in interrupt service routine for capture driver*

Function	void VCAP_isr();
Arguments	none
Return Value	none
Description	<p>This is the built-in interrupt service routine for the capture driver. The user should never call this function directly. It is exported globally so that it can be referenced in the interrupt service table.</p> <p>When using DSP/BIOS, HWI EXTINT5 must be set to call this function and the "Use Dispatcher" box MUST be checked.</p>

VCAP_open *Opens capture device*

Function	int VCAP_open();
Arguments	none
Return Value	<p>success Returns 0 on failure. Failure could happen if required system resources could not be allocated.</p>
Description	<p>This function opens the capture device and must be called before calling any other capture APIs. Generally, this function is called only once at the beginning of your program. VCAP_reset() is called and system resources are allocated. To free up these resources, call VCAP_close().</p> <p>Here are the steps taken.</p> <ul style="list-style-type: none"> <input type="checkbox"/> Check to make sure the capture is not already open <input type="checkbox"/> Dynamically create a DSP/BIOS semaphore <input type="checkbox"/> Call VCAP_reset() <input type="checkbox"/> Returns

Example `success = VCAP_open();`

VCAP_reset *Resets capture hardware*

Function	int VCAP_reset();
Arguments	none
Return Value	none

VCAP_settings

Description This function resets the capture hardware which in effect turns it off. Here are the steps taken:

- Check to make sure capture is open
- Clears all FPGA capture registers
- Disables and clears the capture CPU interrupt
- Updates VCAP_settings
- Returns

Example `VCAP_reset();`

VCAP_settings *Display Code at Selected Address*

Global Variable `extern far VCAP_Settings VCAP_settings;`

Members

<code>mode</code>	Current mode, i.e. VCAP_NTSC
<code>hres</code>	Horizontal resolution in pixels, i.e. 640
<code>vres</code>	Vertical resolution in pixels, i.e. 480
<code>fps</code>	Capture rate in frames per second, i.e. 30

Description This is a global variable exported out of the VCAP module that contains information about the current capture mode. It's purpose is to make it easy for the application code to read this information. It's useful when the application code needs react differently depending on capture settings.

Note: Do **not** write to any members of this structure, it is intended to be read-only.

This structured is altered by `VCAP_open()`, `VCAP_config()`, and `VCAP_reset()`. Generally, the user calls `VCAP_config()` then uses information from this structure.

Example `VCAP_config(VDIS_NTSC);`

```
dx = VCAP_settings.hres;  
dy = VCAP_settings.vres;
```

Examples

The examples provided in this section are meant to be illustrative examples of Video Device Drivers use and do not necessarily provide the most performance optimized means of implementing the example scenarios. Please refer to the *TMS320C6000 Imaging Developer's Kit (IDK) Programmer's Guide* (Literature number SPRU495) for further information on performance optimized implementations of video capture and display.

All of the examples can assume this for a *main* function.

```

/*-----*/
void main() {
    /* initialize the CSL library */
    CSL_init();

    /* open a DMA channel for the DAT module */
    DAT_open(DAT_CHAANY, DAT_PRI_LOW, DAT_OPEN_2D);
    /* open up the video systems */
    VDIS_open();
    VCAP_open();

    /* Remember that the main task 'tskMain' will execute automatically */
    /* once we exit 'main' and DSP/BIOS starts. */
}
/*-----*/

```

Topic	Page
4.1 Draw a Box in 640x480x16 Display Mode	4.1
4.2 Draw a Box in 640x480x8 Display Mode	4.2
4.3 NTSC Capture to 640x480x16 Display Loopback	4.3
4.4 NTSC Capture to 640x480x8 Display Loopback	4.4

4.1 Draw a Box in 640x480x16 Display Mode

```
/*-----*/
void tskMainFunc() {
    Uint16 *d;
    int x,y,dx,dy;
    int frameCnt;
    /* configure the display hardware */
    VDIS_config(VDIS_640X480X16);
    /* get information about the display settings */
    dx = VDIS_settings.hres;
    dy = VDIS_settings.vres;
    /* let's go around for three display frames to ensure we */
    /* render all three display buffers */
    for (frameCnt=0; frameCnt<3; frameCnt++) {
        /* grab the next available display buffer */
        d = (Uint16*)VDIS_toggleBufs(SYS_FOREVER);
        /* fill display with solid color */
        for (y=0; y<dy; y++) {for (x=0; x<dx; x++) {d[dx*y+x] = 0x001F;}}
        CACHE_flush(CACHE_L2ALL,0,0);
        /* draw a box around perimeter of display */
        for (x=0; x<dx; x++) d[dx*0+x] = 0xF800; /* top */
        for (x=0; x<dx; x++) d[dx*(dy-1)+x] = 0xF800; /* bottom */
        for (y=0; y<dy; y++) d[dx*y+0] = 0xF800; /* left */
        for (y=0; y<dy; y++) d[dx*y+(dx-1)] = 0xF800; /* right */
        CACHE_flush(CACHE_L2ALL,0,0);
    }
    /* loop forever */
    while (1) {
        VDIS_toggleBufs(SYS_FOREVER);
    }
}
/*-----*/
```

4.2 Draw a Box in 640x480x8 Display Mode

```

/*-----*/
void tskMainFunc() {
    Uint8 *d;
    int x,y,dx,dy;
    int frameCnt;
    /* configure the display hardware */
    VDIS_config(VDIS_640X480X8);
    /* get information about the display settings */
    dx = VDIS_settings.hres;
    dy = VDIS_settings.vres;
    /* let's go around for three display frames to ensure we */
    /* render all three display buffers */
    for (frameCnt=0; frameCnt<3; frameCnt++) {
        /* grab the next available display buffer */
        d = (Uint8*)VDIS_toggleBufs(SYS_FOREVER);
        /* fill display with solid color */
        for (y=0; y<dy; y++) {for (x=0; x<dx; x++) {d[dx*y+x] = 0x80;}}
        CACHE_flush(CACHE_L2ALL,0,0);
        /* draw a box around perimeter of display */
        for (x=0; x<dx; x++) d[dx*0+x] = 0xFF; /* top */
        for (x=0; x<dx; x++) d[dx*(dy-1)+x] = 0xFF; /* bottom */
        for (y=0; y<dy; y++) d[dx*y+0] = 0xFF; /* left */
        for (y=0; y<dy; y++) d[dx*y+(dx-1)] = 0xFF; /* right */
        CACHE_flush(CACHE_L2ALL,0,0);
    }
    /* loop forever */
    while (1) {
        VDIS_toggleBufs(SYS_FOREVER);
    }
}
/*-----*/

```

4.3 NTSC Capture to 640x480x16 Display Loopback

```
/*-----*/
void tskMainFunc() {
    VCAP_Frame *input;
    Uint16 *output;
    int frameCnt,line;
    int ddx,ddy,cdx,cdy;
    Uint8 *y1,*cr1,*cb1,*y2,*cr2,*cb2;
    VDIS_config(VDIS_640X480X16);
    VCAP_config(VCAP_NTSC);

    ddx = VDIS_settings.hres;
    ddy = VDIS_settings.vres;
    cdx = VCAP_settings.hres;
    cdy = VCAP_settings.vres;

    /* loop forever */
    while (1) {
        /* get video buffers */
        input = VCAP_getFrame(SYS_FOREVER);    /* synchronize to the capture system
*/
        output = (Uint16*)VDIS_toggleBufs(0);

        y1 = (Uint8*)(input->y1);
        cr1 = (Uint8*)(input->cr1);
        cb1 = (Uint8*)(input->cb1);
        y2 = (Uint8*)(input->y2);
        cr2 = (Uint8*)(input->cr2);
        cb2 = (Uint8*)(input->cb2);
        /* grab frame */
        for (line=0; line<(cdy/2); line++) {
            /* copy odd field over to working buffer */
            DAT_copy(&y1[cdx*line], &yBuff[2*cdx*line], cdx);
            DAT_copy(&cr1[cdx*line/2], &crBuff[cdx*line], cdx/2);
            DAT_copy(&cb1[cdx*line/2], &cbBuff[cdx*line], cdx/2);
        }
    }
}
```



```
/* copy even field over to working buffer */
DAT_copy(&y2[cdx*line],    &yBuff[2*cdx*line+cdx],  cdx);
DAT_copy(&cr2[cdx*line/2], &crBuff[cdx*line+cdx/2], cdx/2);
DAT_copy(&cb2[cdx*line/2], &cbBuff[cdx*line+cdx/2], cdx/2);
}
DAT_wait(DAT_XFRID_WAITALL);

/* do color space conversion */
for (line=0; line<(cdy-3); line++) {
    ybcr422p1_to_rgb565_asm(
        coeffs,
        &yBuff[cdx*line],
        &crBuff[(cdx/2)*line],
        &cbBuff[(cdx/2)*line],
        &output[ddx*(ddy-cdy)/2+(ddx-cdx)/2+ddx*line],
        cdx
    );
}
/* flush the cache */
CACHE_flush(CACHE_L2ALL,0,0);
}
}
/*-----*/
```

4.4 NTSC Capture to 640x480x8 Display Loopback

```
/*-----*/
void tskMainFunc() {
    VCAP_Frame *input;
    Uint8 *output;
    int frameCnt,line;
    int ddx,ddy,cdx,cdy;
    Uint8 *y1,*y2;
    VDIS_config(VDIS_640X480X8);
    VCAP_config(VCAP_NTSC);

    ddx = VDIS_settings.hres;
    ddy = VDIS_settings.vres;
    cdx = VCAP_settings.hres;
    cdy = VCAP_settings.vres;

    /* loop forever */
    while (1) {
        /* get video buffers */
        input = VCAP_getFrame(SYS_FOREVER);
        output = (Uint8*)VDIS_toggleBufs(0);

        y1 = (Uint8*)(input->y1);
        y2 = (Uint8*)(input->y2);
        /* grab frame */
        for (line=0; line<(cdy/2); line++) {
            /* copy odd field, Y only */
            DAT_copy(&y1[cdx*line], &yBuff[2*cdx*line], cdx);

            /* copy even field, Y only */
            DAT_copy(&y2[cdx*line], &yBuff[2*cdx*line+cdx], cdx);
        }
        DAT_wait(DAT_XFRID_WAITALL);

        /* copy Y data to output display buffer */
    }
}
```

```
    for (line=0; line<cdy-3; line++) {
        DAT_copy(&yBuff[cdx*line], &output[ddx*(ddy-cdy)/2+(ddx-cdx)/2+ddx*line],
cdx);
    }
    DAT_wait(DAT_XFRID_WAITALL);

    /* flush the cache */
    CACHE_flush(CACHE_L2ALL,0,0);
}

/*-----*/
```